Fall 2019 CSE 534 Project Report
Adaptive Video Streaming

David Paredes-Merino (112280378), Zekun Zhang (112070717)
{dparedesmeri,zekzhang}@cs.stonybrook.edu

December 6, 2019

1 Problem Description

1.1 Adaptive Video Streaming

Adaptive video streaming is critical for video websites to provide smooth and high-quality videos streaming experience to their customers. In a typical setup, the server encodes the video with different bitrate settings, then split each of the output videos into short chunks. The video streaming client fetches those chunks and puts them into its local buffer, then the player plays the chunk to the viewer. At each boundary between two chunks, the client needs to select among different bitrates according to current network condition. When the bandwidth is limited, it fetches low bitrate chunks to make sure the buffer is not drained so viewer will not have a pause. When network condition is good, it fetches high bitrate chunks to have high-quality video. The overall goal is to provide highest quality video without smoothness being compromised.

It is not trivial to develop efficient adaptive bitrate (ABR) selection algorithms, because the network condition is highly varied, and the choice of selection has long-term consequences. The algorithm needs to predict what the network condition will be in the near future, then make decision based on such predictions and the number of chunks in the buffer. Also the metrics to evaluate the performance are conflicting, some of them value video quality while others prefer smoothness. Most adaptive video streaming clients use hand-crafted bitrate selection functions. These methods perform well under designated conditions, but can fail under unseen networks. Also each of them is usually fixed to improving only one certain user-experience metric.

1.2 Streaming Algorithms

In this project, the experiment environment we use is GPAC[2]. There are 4 adaptive streaming algorithms implemented in GPAC: simple buffer-size based algorithm, BBA[1], BOLA-basic and BOLA-finite [6].

Buffer-based methods selects the bitrates as a function of buffer occupancy. For the case of simple buffer-based method, it sets a minimum and maximum threshold for the video chunks buffer. These values are usually 1/3 and 2/3 of the total size of the buffer size respectively. When the total length of the chunks in the buffer is higher than the maximum threshold, it selects higher-bitrate chunks to fill the buffer. In the case where the total length of the chunks is in between both thresholds, the bitrate is not changed. Otherwise it decreases the quality of next video chunk.

BBA is a type of buffer-size based. The key difference is that this paper introduces the concept of "reservoir". First, we make sure to fill up the reservoir with the minimum bitrate. Then, we can increase the bitrate linearly with respect to buffer occupancy. In the case where the traffic slows down and the buffer might dry before downloading another chunk of video, the bitrate sets to minimum and fills up again the reservoir. In other words, in order to avoid any interruption while streaming a video, the buffer occupancy should be at least the size of the reservoir. It is important to mention that in the GPAC implementation, the reservoir size is 37.5% of the buffer maximum size.

BOLA is another buffer-size based algorithm that uses Lyapunov optimization to choose the bitrates and maximizes a quality of experience metric(QoE) and takes into account previous buffer occupancy observations. We work on two specific types: BOLA basic and BOLA finite. BOLA basic assumes a condition that video streaming has infinite length for its derivation. BOLA finite targets smaller videos and uses wind-up and down phases to be more cautious, it means that it does not try to fill up the buffer too soon and does not try to keep the buffer full too long.

1.3 Proposed Methods

In this project we will try different methods to evaluate the performance of bitrate selection algorithms. Based on the observation, we try to compare different algorithms under different network conditions.

It is also possible to apply machine learning techniques into this problem, since machine learning models can generalize well if trained properly. If we consider the network condition and buffer status as the environment, the stream client as the agent, the bitrate selection operation as the action of the agent, and the cumulative QoE performance as the reward, the ABR problem can be regarded as a reinforcement learning problem. There are numerous ways to model the problem, we will be using Q-learning. Formally, for a given state of the environment s, we try to learn a value function $q(s, a)$, which gives the expectation of reward r if the agent takes action a at this state. When this value function is learned, the policy π of the agent will be to take the action of the highest value

$$\pi(s) = \arg\max_a q(s, a). \tag{1}$$

The value function is learned through iterations of value update. At time t the state is s_t, the agent takes action based on current value function. After the action is taken, the state changes to s_{t+1}, and reward r_t is observed. The value function is then updated following

$$q(s_t, a_t) \leftarrow (1 - \eta)q(s_t, a_t) + \eta[r_t + \gamma \max_a q(s_{t+1}, a)], \tag{2}$$

where η is the learning rate, and γ is the discount factor, both in range $(0, 1)$. When the number of possible states and actions is limited, it is possible to estimate the value function as a lookup table. However, when the space of state or action is continuous, q also has to be a continuous function of s and a. If the value function takes the form of a deep neural network, it is referred as deep Q-network (DQN)[4]. Here the value function is a deep neural network $Q(s, a; \mathbf{w})$, where $\mathbf{w}$ is the weights of the network. Accordingly, the network is updated using gradient descent

$$\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla_{\mathbf{w}}[Q(s_t, a_t; \mathbf{w}) - r_t - \gamma \max_a Q(s_{t+1}, a; \mathbf{w})]^2. \tag{3}$$

Here η and γ are still learning rate and discount factor. $\mathbf{w}$ is optimized so the estimated reward $Q(s_t, a_t; \mathbf{w})$ matches $r_t + \gamma \max_a Q(s_{t+1}, a; \mathbf{w})$.

DQN is not easy to train. And an important part of the training process is the tradeoff between exploitation and exploration. If the agent always follows the current policy during training when taking actions, it will likely be trapped in a local minimum value. So instead of greedily following the action that has the highest value, at each time step, the agent will take a random action with probability ϵ. With probability $1 - \epsilon$, it takes the current optimal action. This is called ϵ-greedy, and the value of ϵ usually decays over training episodes.

2 Implementation Details

Our code of implementation at GitHub: https://github.com/zvant/gpac_ABS.

2.1 Adaptive Video Streaming Environment Setup

We have setup our experiment environment using GPAC[2] package. It provides `MP4Box`, which can be used to prepared the video chunks for adaptive streaming; and `mp4client`, which includes fully-functional MPEG-DASH functionalities with several bitrate selection algorithms. GPAC is compiled on both Linux server and Windows desktop systems.

On the server side, firstly the source video is encoded by the x264 codec into different resolutions (and therefore different bitrates) using `FFmpeg`. A stamp that contains the current playback time and the resolution is also added to each video frame, making it easier to run the experiments. The audio track is stripped to simplify the problem. For example, the command for encoding the 720P version of the video is

```
ffmpeg -i input.mp4 -an -vcodec libx264 -vf "drawtext=text='720P
↪  %{eif\:t\:d}':fontcolor=white:
↪  fontsize=24:box=1:boxcolor=black:boxborderw=5:x=10:y=h-34,
↪  scale=1280:720" 720P_text.mp4
```

The stamped videos are then re-encoded using specified x264 parameters. This step is important, since the desired bitrate, interval between key frames, and encoding preferences are set. Note that if the videos are going to be split into chunks that last S seconds, the interval between key frames needs to be set less than S. Otherwise the chunks will be corrupted. A sample command is

```
x264 --output 720P.264 --fps 24 --preset slow --bitrate 2000 --vbv-maxrate
↪  4000 --vbv-bufsize 8000 --min-keyint 48 --keyint 48 --scenecut 0
↪  --no-scenecut --pass 1 720P_text.mp4
MP4Box -add 720P.264 -fps 24 720P.mp4
```

The generated video files are then split into small chunks using `MP4Box`. The chunks from each video is regarded as a track. This process also generates a media presentation description (MPD) file, which is an XML-based manifest file that contains the length of the whole video and information of each track. The information includes track ID, data type, codec, resolution, and bitrate. The command used is

```
MP4Box -dash 4000 -rap -frag-rap -profile live -out dash.mpd
↪  900P.mp4#video 720P.mp4#video 450P.mp4#video 360P.mp4#video
```

The MPD file, along with all the chunks, is served over HTTP by a typical HTTP server program. In our setup NginX is used, since it is a light-weighted fully-functional HTTP server program.

The `mp4client` first reads the information in the MPD file, and initialize the video player. Then it decides which track to select for the next video chunk based on the current

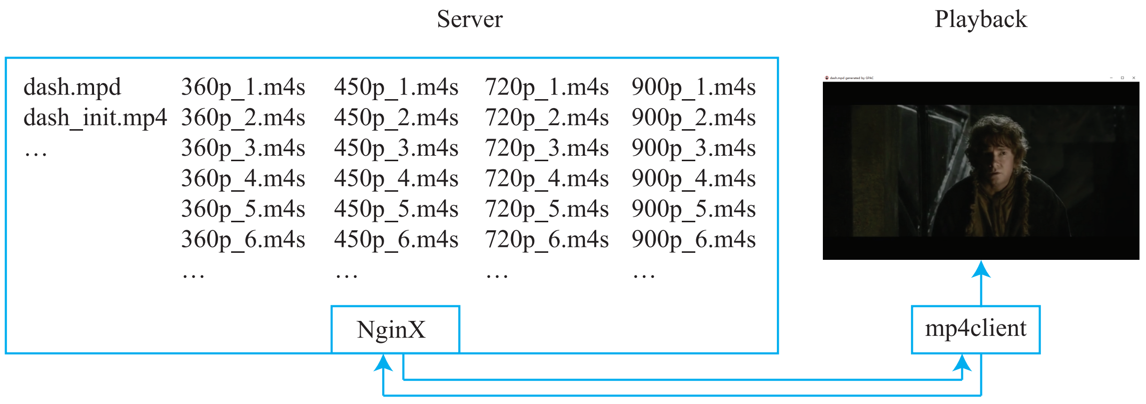

Figure 1 The Server-Client Architecture of the Setup

bandwidth estimation and buffer status, until reaches the end of the whole video. The client-server architecture of our setup is illustrated in Figure 1.

We use `tc` command to simulate different network conditions on the server side. For example, to set the loss rate on `eth0` to 6%, the command is

```
sudo tc qdisc add dev eth0 root netem loss 6%
```

To restrict the bandwidth to 10 kbps and the latency to 100 ms, the command is

```
sudo tc qdisc add dev eth0 root netem rate 10kbit burst 5kbit latency
 ↪ 100ms
```

2.2 QoE Metrics Measurement

Quality of experience(QoE) is formulated as the sum of bitrates of all video chunks and it is penalized by two factors. The first one is rebuffering, which is the time between downloading and playing out the video chunk. The second factor is the absolute difference between consecutive bitrates. Thus, a generic QoE can be defined as a linear combination as follows

$$QoE = \sum_{n=1}^{N} q(R_n) - \mu \sum_{n=1}^{N} T_n - \sum_{n=1}^{N-1} |q(R_{n+1}) - q(R_n)|, \tag{4}$$

where N is the number of video chunks, R_n is the bitrate in kbps of the n-th chunk, and $q(R_n)$ is a mapping that depends on the quality perceived. For simplicity, in our case we use $q(R_n) = R_n$. T_n is the rebuffering time in ms, and the factor is set as $\mu = 4.3$ as explained in [3].

2.3 DQN Trainer and Inference Client

The state s is defined as the status of the estimated network throughput (bps), bitrate of the last chunk (bps), and length of the buffer (ms) of the past 5 time steps. Each of the values are divided by 10×6, 10×6, and 10×4 to have a comparable scale. So the input of the DQN is a 15-dimensional vector. The output of the DQN is a 4-dimensional vector, each dimension represents the value of one of the 4 bitrates. The network has 3 fully-connected layers, the layers have the shapes of 15×32, 32×32, and 32×4, respectively. The 1st and 2nd layers are followed by ReLU activation.

The observed reward at each iteration is a weighted sum of the video quality and smoothness of streaming. The video quality r_q is represented by the bitrate (bps) of the next chunk divided by 10^7. The smoothness r_s is represented by the buffer size (ms) divided by 10^4, since longer buffer size indicates that it is less likely the streaming will be paused. For a set weight α, the reward is

$$r = \alpha r_q + (1 - \alpha) r_s. \tag{5}$$

The DQN model is trained with different values of $\alpha = 0.0, 0.5, 0.9, 1.0$ to compare.

DQN is trained in an off-policy manner. The weights are updated after each training episode. The model is trained for 100 episodes, and each training episode is the process of 120 seconds of video streaming, which is 30 video chunks. During each episode, the policy is fixed, and the client selects bitrate using ϵ-greedy according to current policy. After the episode is finished, the weights of DQN are updated using gradient descent. The training process is summarized in algorithm 1.

Algorithm 1: DQN Training Algorithm

```
random initialize DQN $Q(s, a; \mathbf{w})$;
train for $E$ episodes, each episode contains $T$ time steps;
set $\eta = 5 \times 10^{-4}$, $\gamma = 0.85$;
for $ep = 1 \dots E$ do
    initialize transactions set $D = \{\}$;
    set $\epsilon = 0.95^{ep}$;
    for $t = 1 \dots T$ do
        get current state $s_t$;
        select bitrate of next chunk using $\epsilon$-greedy, get action as $a_t$;
        get next state $s_{t+1}$ and reward $r_t$;
        $D = D \bigcup \{[s_t, a_t, s_{t+1}, r_t]\}$
    end
    for $[s_t, a_t, s_{t+1}, r_t]$ in $D$ do
        $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} [Q(s_t, a_t; \mathbf{w}) - r_t - \gamma \max_a Q(s_{t+1}, a; \mathbf{w})]^2$;
    end
end
```

The training of the DQN is implemented in PyTorch[5], while the inference of the network is directly implemented in GPAC using matrix multiplication. At each streaming episode, the weights of the network is exported into a text file. Then the GPAC client reads the weights and do the DQN inference. It also stores the transactions into another text file, which is then read by the training program to update the weights. The main functionalities are implemented in the functions `dash_rl_read_model`, `rl_inference`, `rl_build_state`, `rl_add_state`, `rl_get_state`, and `dash_do_rate_adaptation_rl` located in the source file `src/media_tools/dash_client.c`. The training program is located in `DASH_rl_train` directory.

To simulate different network conditions, `tc` command is used to restrict the network throughput. On the server side, different levels of packet loss rates λ are used. When not restricted, the network is regarded as unlimited. 6% loss rate is used for simulating moderate packet loss. And 16% loss rate is used to simulate highly congested network conditions. The model is trained with different loss rates to compare its behaviour under different network conditions.

3 Experiment Results and Discussions

3.1 Streaming Algorithms Comparison

We use the 4 streaming algorithms and a sample video of 506 seconds total, which is stored in the server side at 4 video resolutions: 640×360, 854×480, 1280×720 and 1600×900 and their bitrates levels are: 645 Kbps, 1030 Kbps, 1877 Kbps and 2701 Kbps respectively. The length of each video chunk is 12 seconds and 24 frames per second.

In the first network setting, we force the server network to drop a fixed percentage of packets from 2% until 10% and we do not change any other parameter. Also, the traffic network is not limited. The results are depicted in Figure 2 and we can observed that regardless of the method, the normalized QoE drops drastically after around 6%. Also, BOLA does a better job because its Lyapunov optimization targets directly the video quality as well as minimizing rebuffering time.

In the second network setting, we limit the maximum value of the simulated bandwidth throttling as seen in the top plot of Figure 3 and it changes every 60 seconds. The idea is to explore how the streaming algorithms change the bitrate selection and the buffer occupancy over time. In the initial minutes when there is a slow traffic network, BOLA does suffer rebuffering. BBA instead performs better because it always cares about filling up the "reservoir" (fraction of the total buffer) with the lowest bitrate and that quality is enough for this slow network traffic. However, in the next minutes when the server have a higher network traffic, BOLA performs better because it can download video chunks with high bitrates and the buffer has always content to play. In the early and later minutes, simple buffer-based method does a decent job because it does not suffer of rebuffering time but it sacrifices the bitrate quality.

In summary, given this particular network setting of low packet loss rate, BOLA perform better than the rest. Otherwise, no other method works well. When we have control of the bandwidth throttling, in low network traffic conditions BOLA does not work properly because of rebuffering. On the other hand, BBA can reach high bitrates but in longer time.

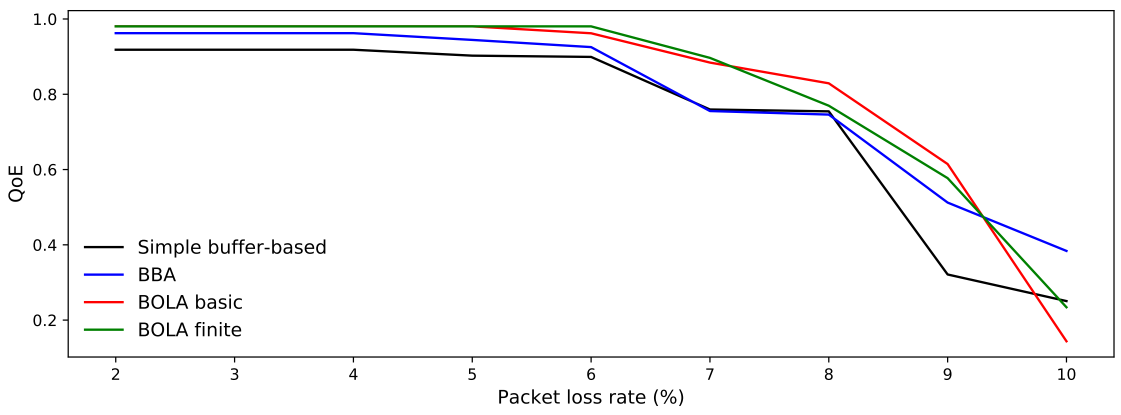

Figure 2 Normalized QoE for different fixed packet losses rates

3.2 DQN Behaviour

In our setting, we use 4 different video resolutions: 640×360, 800×450, 1280×720, and 1600×900. Their bitrates are 776 Kbps, 971 Kbps, 1944 Kbps, and 2724 Kbps, respectively. And the duration of the video chunks is set to 4 seconds. After training the

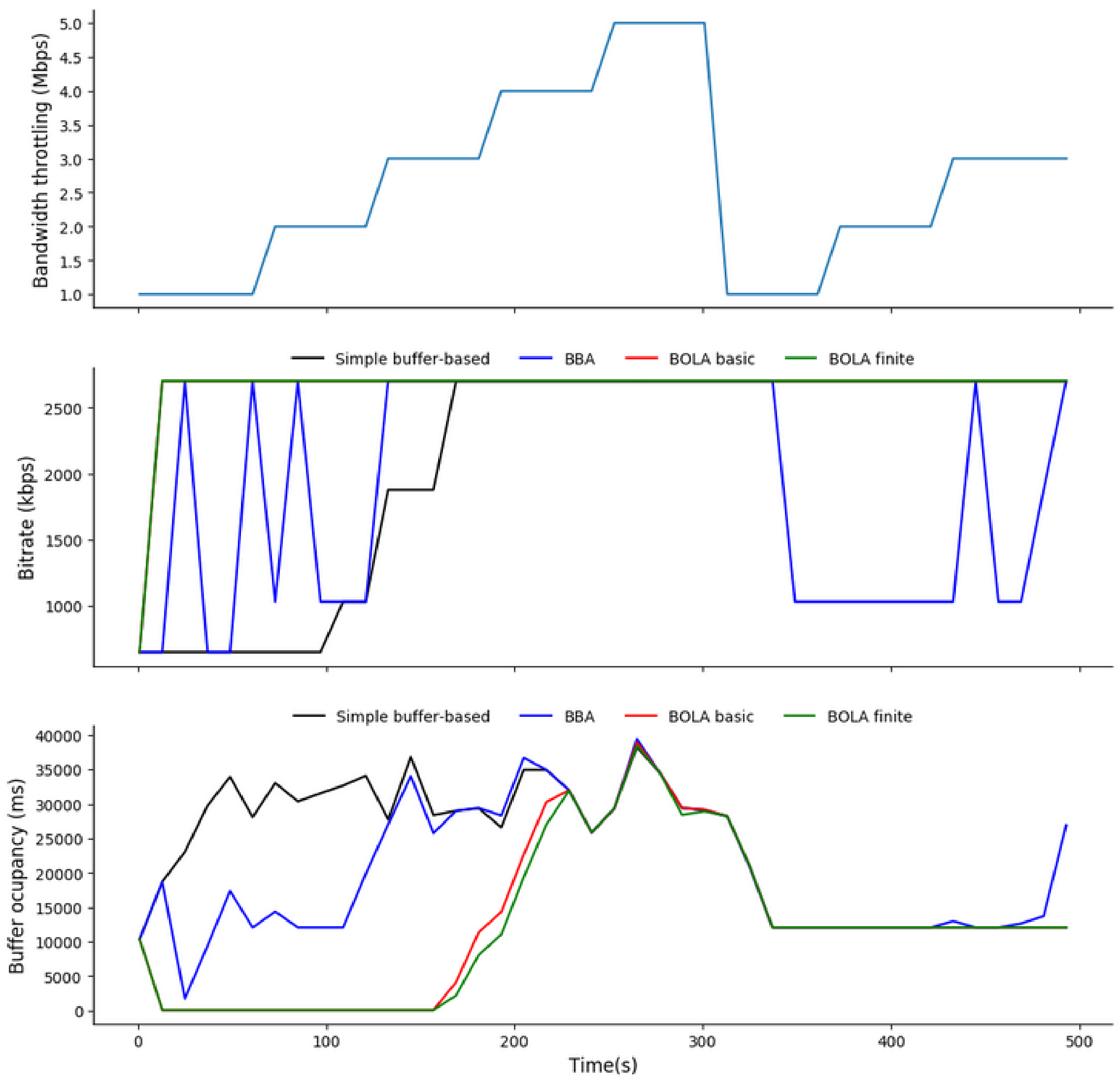

Figure 3 Evolution of bitrate selection and buffer size over time in a simulated bandwidth throttling of the server network

DQN for 100 episodes, the model is tested in the same network condition as training time. The client select the bitrate of next video chunk greedily based on the learned DQN (e.g. $\epsilon = 0$). We define the behaviour of the DQN model as the bitrate selection of the first 20 chunks. The quality of streaming q is defined as the averaged bitrate of fetched chunks. The smoothness of streaming s is defined as the seconds that the video is paused. This is calculated from the time taken to fetch 20 chunks minus 80 seconds. If this time negative, there is no pause.

Table 1 shows the seconds of pause under different network conditions with models trained with different reward weights. When the loss rate is lower than 6%, there is no pause. When loss rate is 16%, the pause times are very varied and do not have a clear trend. Table 2 shows the average bitrate under different network conditions with models trained with different reward weights. When the network throughput is unlimited, it is clear that model trained with larger α tends to select higher bitrate chunks. However, when the network is congested, this trend becomes unclear again, especially when loss rate is 16%. The uncertain trends under highly congested network conditions might be caused by the fact that high loss rate introduces much uncertainty to the training and testing process, thus makes the result hard to interpret.

		α			
		0.0	0.5	0.9	1.0
λ	0%	0	0	0	0
	6%	0	0	0	0
	16%	80	5	111	65

Table 1 Streaming Smoothness (s) of Different Reward Weight α and Loss Rate λ

		α			
		0.0	0.5	0.9	1.0
λ	0%	776	971	1944	2626
	6%	2626	1058	1497	1789
	16%	2237	961	2626	1876

Table 2 Streaming Quality (kbps) of Different Reward Weight α and Loss Rate λ

The selection behaviour of different DQN models is illustrated in Figure 4. When there is no loss, models trained with different reward weight 0.0, 0.5, 0.9, and 1.0 constantly select chunks of different bitrates, which is consistent with expectation. When loss rate is 6%, the model trained with $\alpha = 0.5$ still selects 1000 kbps chunks. However, model trained with $\alpha = 0.0$ always selects highest bitrate chunks, which is not consistent with the meaning of α. Models with $\alpha = 0.9$ and $\alpha = 1.0$ are very unstable, oscillating between highest and lowest bitrates. At 16% loss rate, the behaviour of model trained with $\alpha = 0.5$ is still the same. But model trained with $\alpha = 0.9$ selects the highest bitrate chunks, while other 2 models are not stable.

Both the policy evaluation and the behaviour analysis indicate that under highly congested network, the DQN model fails to converge. This might be caused by the fact that congestion introduces much uncertainty, making the DQN model unstable during training.

3.3 Future Work

Our experiments of comparing different DASH algorithms are only limited to very small number of network conditions. It requires more experiments to explore the performance of the algorithms to form a more solid conclusion. Also the DQN-based method should be compared with the hand-crafted algorithms using the same QoE metrics to evaluate its performance.

References

[1] Huang, T.-Y., Johari, R., McKeown, N., Trunnell, M., and Watson, M. A buffer-based approach to rate adaptation: Evidence from a large video streaming service. In *SIGCOMM '14* (2014).

[2] Le Feuvre, J., Concolato, C., and Moissinac, J.-C. GPAC: open source multimedia framework. In *MM '07* (2007).

[3] Mao, H., Netravali, R., and Alizadeh, M. Neural adaptive video streaming with pensieve. In *SIGCOMM '17* (2017).

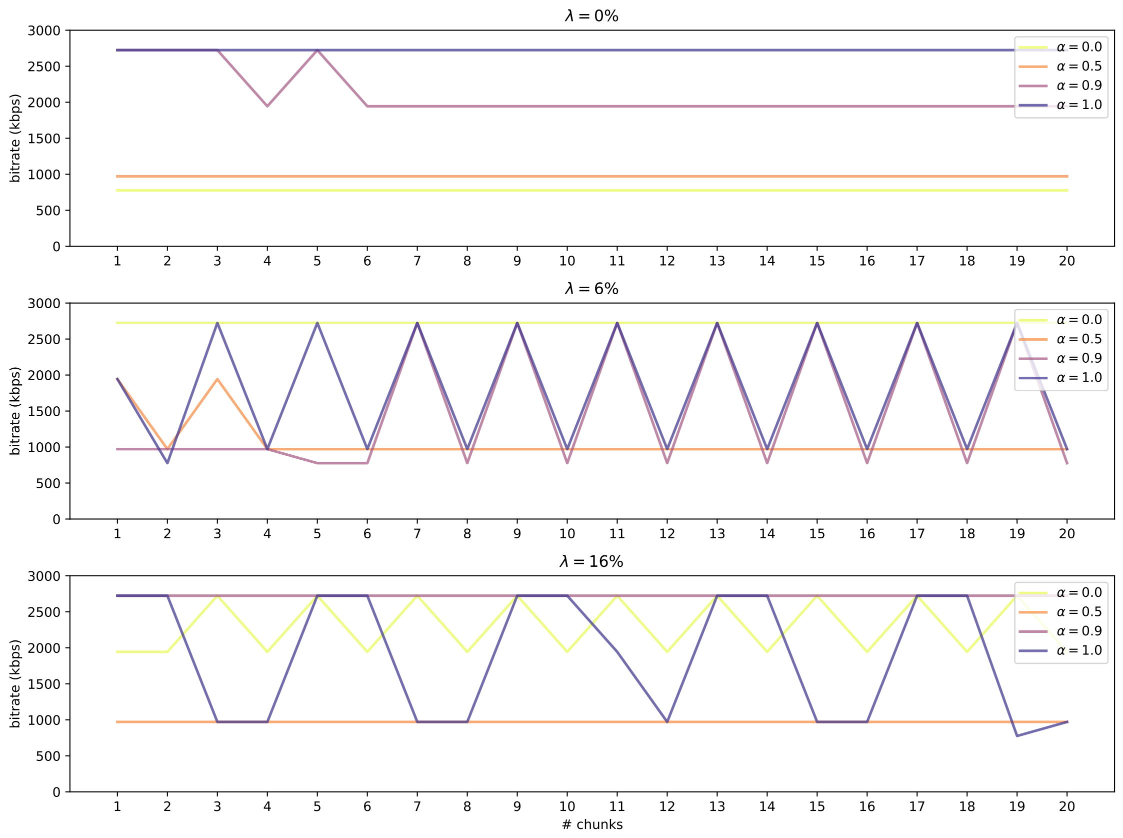

Figure 4 Bitrate Selection Behaviour of Different Reward Weight α and Loss Rate λ

[4] Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. Playing atari with deep reinforcement learning. In *NIPS Deep Learning Workshop*. Curran Associates, Inc., 2013.

[5] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. Automatic differentiation in PyTorch. In *NeurIPS Autodiff Workshop* (2017), Curran Associates, Inc.

[6] Spiteri, K., Urgaonkar, R., and Sitaraman, R. K. BOLA: near-optimal bitrate adaptation for online videos. In *INFOCOM '16* (2016).